TRABAJO PRACTICO FINAL POO2

“A LA CAZA DE LAS VINCHUCAS”

AÑO 2022

INTEGRANTES:

Lucas Alvarez Garcia (lucasss162@gmail.com)

Lucía Sciacca (luly_sciacca@hotmail.com)

Luciano Ruggeri (lucianoruggeri5@gmail.com)

DECISIONES DE DISEÑO DEL SISTEMA:

La mayoría de las clases implementan interfaces para limitar su protocolo y para que sean fácilmente remplazables.

PATRONES UTILIZADOS EN EL SISTEMA:

Patrón State:

Propósito:

- Permite que un objeto modifique su comportamiento cada vez que cambie su estado interno, parecerá que cambia la clase del objeto.

El patrón State se puede utilizar cuando:

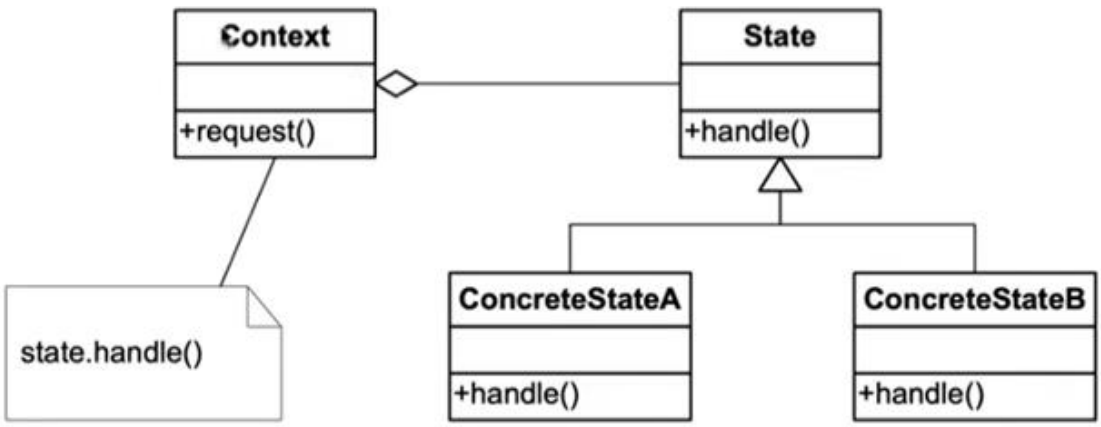
- El comportamiento de un objeto depende de su estado y debe cambiar en tiempo de ejecución dependiendo de ese estado.

- Las operaciones tienen largas sentencias condicionales con múltiples ramas que dependen del estado del objeto. El patrón State pone a cada rama de la condición en una clase aparte. Esto nos permite tratar al estado del objeto como un objeto que puede variar independientemente de otros objetos.

Pros y contras:

- ✓ Principio de responsabilidad única. Organiza el código relacionado con estados particulares en clases separadas.
- ✓ Principio open / closed. Introduce nuevos estados sin cambiar de estado existente o la clase contexto.
- ✓ Simplifica el código del contexto eliminando voluminosos condicionales de máquina de estados.
- ☒ Contra: Aplicar el patrón puede resultar excesivo si una máquina de estados tiene unos pocos estados o raramente cambia.

Estructura:



Clase Muestra

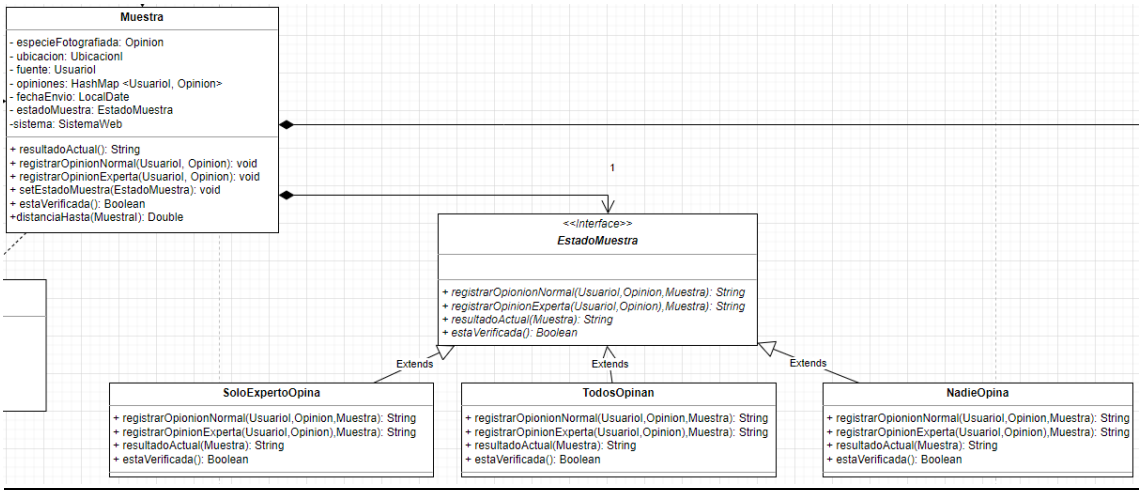
Para modelar la clase muestra empleamos el patrón State para cambiar el comportamiento de la muestra entre los distintos estados que puede tomar.

Los roles distinguidos en este caso son la Muestra la cual mantiene una instancia de un estado concreto, luego tenemos un estado que es representado por la interface EstadoMuestra, la cual define los mensajes que deben entender todos los estados concretos y por últimos los estados concretos que implementan distinto comportamiento según el estado cambia son los siguientes.

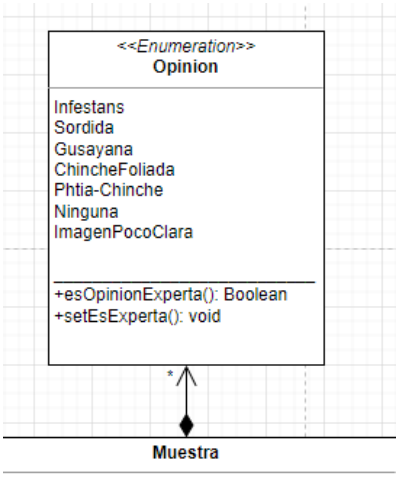
1-OpinionNormal (todos opinan)

2-OpinionExperta (Solo expertos opinan)

3-OpinionVerificada (nadie opina)



Además vale mencionar que para guardar las opiniones decidimos utilizar un Map<UsuarioI, Opinion> para asegurar que un usuario pueda opinar solamente 1 vez sobre cada muestra.
La opinión fue modelada de la siguiente manera



En un principio se diseñó la opinión como una interfaz que era implementada por 2 enumerativos distintos (Vinchuca y NoVinchuca). Esto era para evitar que el usuario cargue

una muestra de manera incorrecta seleccionando una de las siguientes opciones (ninguna, ImagenPocoClara o algún tipo de opción que no fuera vinchuca) como especie fotografiada. Pero luego se combinó ambos enumerativos en uno solo para simplificar el modelo. Y como sugirió un profesor en una de las consultas, se podría evitar que el usuario cargue una Muestra de manera incorrecta utilizando otra capa del sistema por ejemplo alguna validación en el front-end o que el front-end solamente muestre las 3 opciones de vinchuca a la hora de cargar una muestra.

Clase Usuario

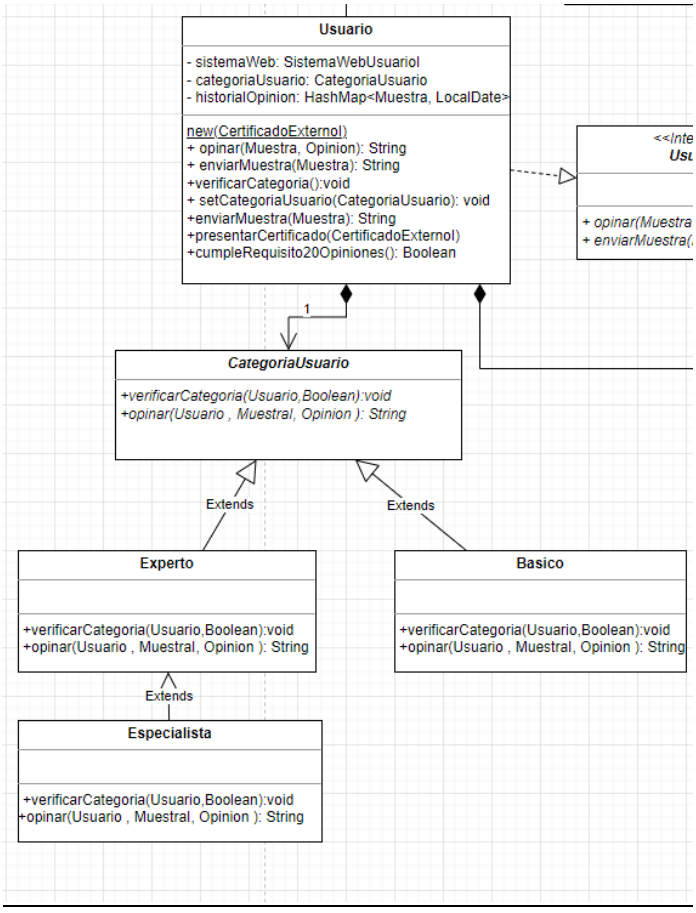
Para modelar la clase usuario también empleamos el patrón State para cambiar el comportamiento del usuario entre los distintos estados que puede tomar. *CategoriaUsuario* se utiliza como un estado para el usuario, ya que se necesita que cambie su comportamiento dependiendo si es experto, un usuario básico o especialista, este último solo se podría utilizar instanciándolo o seteándolo directamente.

Estados Concretos:

1-Basico

2-Experto

3-Especialista: es considerado una subclase de usuario experto porque comparten comportamiento a la hora de opinar sobre una muestra.



Patrón Singleton:

Propósito:

- Garantiza que una clase sólo tenga una instancia y proporciona un punto de acceso global a ella.

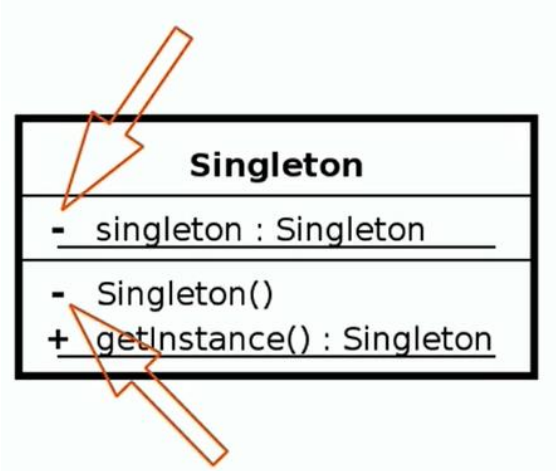
Participantes:

- Define una operación getInstance () que permite que las clases accedan a su única instancia.

- Es responsable de crear su única instancia.

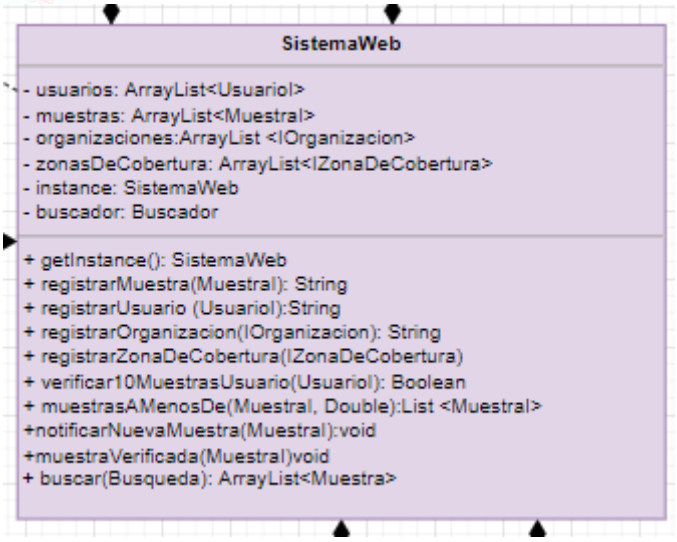
Colaboradores:

-Las clases acceden a la instancia de un Singleton exclusivamente a través de la operación getInstance () de éste.



Clase SistemaWeb:

Es la clase encargada de comunicar y registrar la información entre los distintos componentes del dominio del sistema. Para ello conoce a las 4 clases principales Usuario, Muestra, Organización y ZonaDeCobertura.
El sistema web aplica el patrón Singleton para asegurar que todas las otras clases que lo necesitan conocer solo tengan referencia a una misma instancia.
Para comunicar la información de nuevas muestras o de muestras que fueron validadas a las distintas ZonasDeCobertura incluye los mensajes notificarNuevaMuestra (MuestraI) y muestraVerificada (MuestraI) que itera sobre la lista de ZonasDeCobertura y delega la responsabilidad a cada Zona de identificar si la muestra pertenece o no a su ubicación.
Además vale mencionar que el sistema contiene un colaborador del tipo Buscador al cual le delega la responsabilidad de filtrar las distintas búsquedas pedidas por el enunciado.



Patrón Composite

Propósito:

- Es un patrón de diseño estructural que te permite componer objetos en estructuras de árbol y trabajar con esas estructuras como si fueran objetos individuales.

Participantes:

Componente (Component):
- Declara la interfaz de los objetos de la composición.
- Implementa el comportamiento predeterminado de la interfaz que es común a todas las clases.
- Declara una interfaz para acceder a sus componentes hijos y gestionarlos.
- (opcional) define una interfaz para acceder al padre de un componente en la estructura recursiva y, si es necesario, la implementa.

Hoja (Leaf):
- Representa objetos hoja en la composición. Una hoja no tiene hijos.
- Define el comportamiento de los objetos primitivos de la composición.

Compuesto (Composite):
- Define el comportamiento de los componentes que tienen hijos.
- Almacena componentes hijos.
- Implementa las operaciones de la interfaz Componente relacionadas con los hijos.

Cliente (Client):
- Manipula objetos en la composición a través de la interfaz Componente.

Pros y contras:

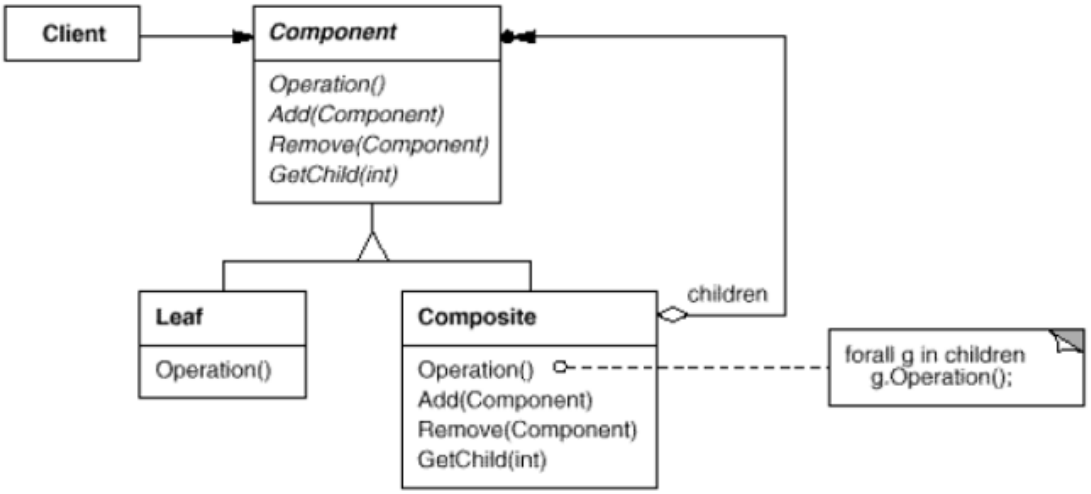
- ✓ Puedes trabajar con estructuras de árbol complejas con mayor comodidad: utiliza el polimorfismo y la recursión en tu favor.

- ✓ Principio de open/closed. Puedes introducir nuevos tipos de elemento en la aplicación sin descomponer el código existente, que ahora funciona con el árbol de objetos.
- ☒ Puede resultar difícil proporcionar una interfaz común para clases cuya funcionalidad difiere demasiado. En algunos casos, tendrás que generalizar en exceso la interfaz componente, provocando que sea más difícil de comprender.

Colaboraciones:

Los Clientes usan la interfaz de la clase Componente para interactuar con los objetos de la estructura compuesta. Si el recipiente es una Hoja, la petición se trata correctamente. Si es un Compuesto, normalmente redirige las peticiones a sus componentes hijos, posiblemente realizando operaciones adicionales antes o después.

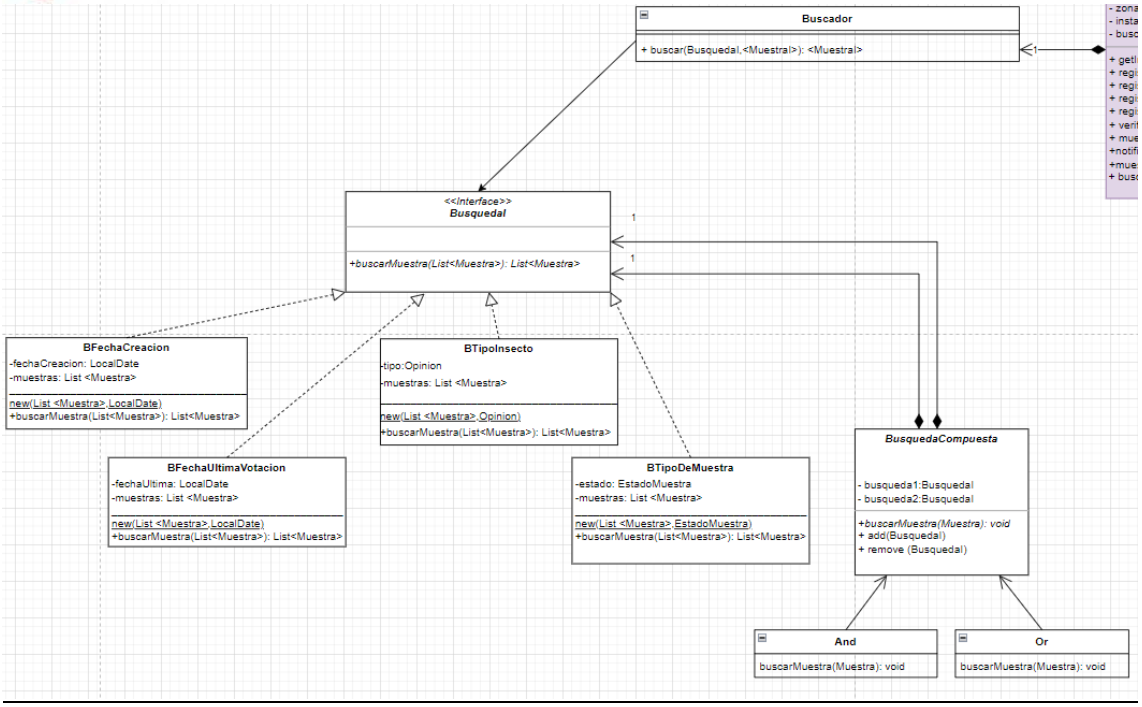
Estructura:



Clase Buscador

La clase buscador es un colaborador del sistema web, al que le delegan la responsabilidad de filtrar las muestras utilizando un tipo de búsqueda específico. En este diseño el cliente seria el Buscador y para modelar las distintas búsquedas utilizamos el patrón Composite donde tenemos una interface BusquedaI aplicando el rol de componente ya que describe la operación buscarMuestra(List<Muestra>) que es comun a elementos simples y complejos del árbol.

En este caso las búsquedas simples como (BFechaCreacion, BFechaUltimaVotacion, BTipoInsecto, BTipoDeMuestra) cumplen el rol de hojas y las búsquedas compuestas formadas por multiples búsquedas simples cumple el rol de composite. A su vez la *BúsquedaCompuesta* es implementada por las clases concretas And y Or, y deja el modelo abierto para agregar en el futuro nuevos operadores de manera simple.



Patrón Observer:

Propósito:

- Define una dependencia de uno-a-muchos entre objetos, de forma que cuando un objeto cambie de estado se notifique y se actualicen automáticamente todos los objetos que dependen de él.

Aplicabilidad:

Usamos el patrón Observer para afrontar cualquiera de las situaciones siguientes:

- ✓ Cuando un cambio en un objeto requiere cambiar otros, y no sabemos cuántos objetos necesitan cambiarse.
- ✓ Cuando un objeto debería ser capaz de notificar a otros sin hacer suposiciones sobre quiénes son dichos objetos. En otras palabras, cuando no queremos que estos objetos estén fuertemente acoplados.

Pros y contras:

- ✓ Principio de abierto/cerrado. Puedes introducir nuevas clases suscriptoras sin tener que cambiar el código de la notificadora (y viceversa si hay una interfaz notificadora).
- ✓ Puedes establecer relaciones entre objetos durante el tiempo de ejecución.
- ☒ Los suscriptores son notificados en un orden aleatorio.

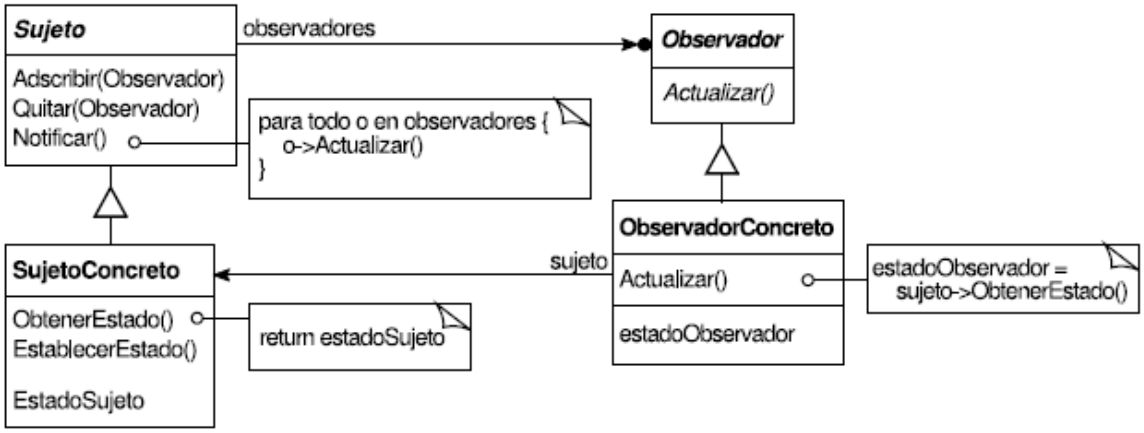
Participantes:

- ✓ **Sujeto:**
  - Conoce a sus observadores. Un sujeto puede ser observado por cualquier número de objetos Observador.
  - Proporciona una interfaz para asignar y quitar objetos Observador.
- ✓ **Observador:**
  - Define una interfaz para actualizar los objetos que deben ser notificados ante cambios en un sujeto.
- ✓ **SujetoConcreto:**
  -Almacena el estado de interés para los objetos ObservadorConcreto.
  - Envía una notificación a sus observadores cuando cambia su estado.
- ✓ **ObservadorConcreto:**
  -Mantiene una referencia a un objeto SujetoConcreto.
  - Guarda un estado que debería ser consistente con el del sujeto.
  - Implementa la interfaz de actualización del Observador para mantener su estado consistente con el del sujeto.

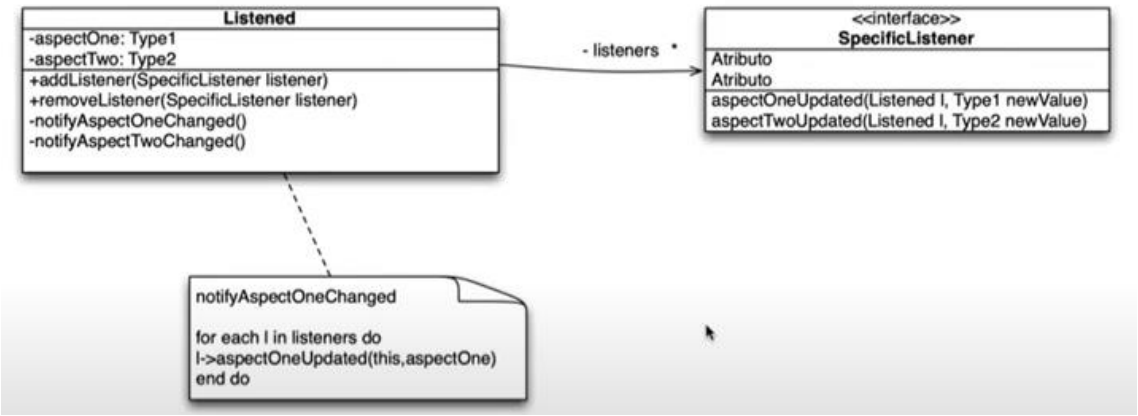
Colaboraciones:

- SujetoConcreto notifica a sus observadores cada vez que se produce un cambio que pudiera hacer que el estado de éstos fuera inconsistente con el suyo.
- Después de ser informado de un cambio en el sujeto concreto, un objeto ObservadorConcreto puede pedirle al sujeto más información. ObservadorConcreto usa esta información para sincronizar su estado con el del sujeto.

Estructura:

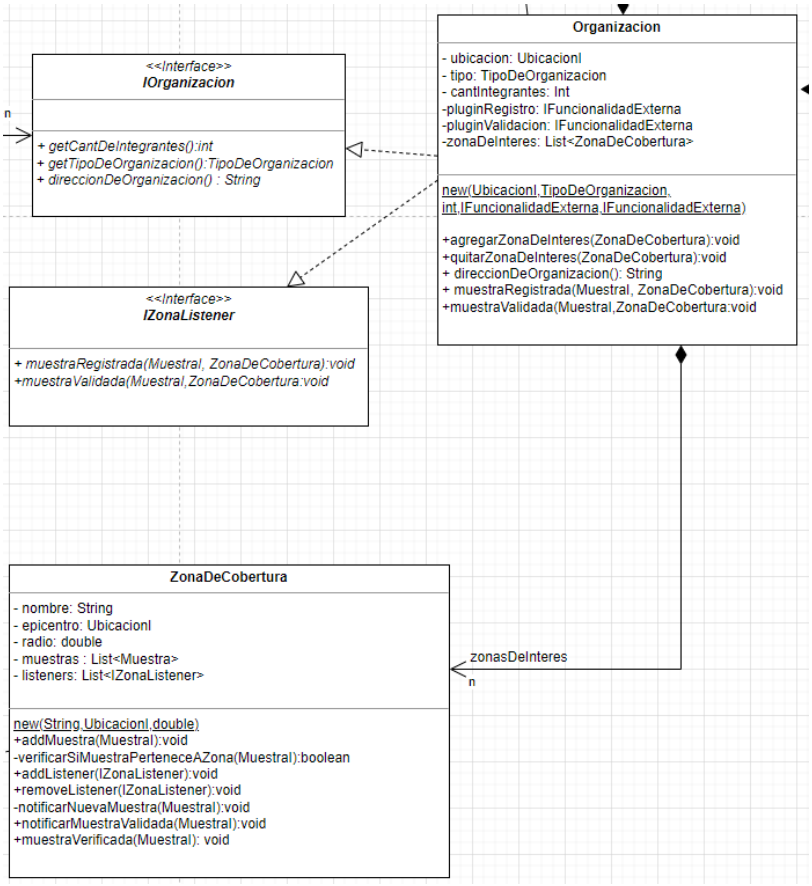


Por cuestiones de diseño decidimos no implementar la interface Observer y Observable de Java y procedimos con el modelo Listeners que aplica los mismos conceptos descritos anteriormente y nos parecía más simple para notificar distintos cambios de estado del objeto observable.



Clase ZonaDeCobertura:

La ZonaDeCobertura cumple el rol de Listener e implementa los métodos addListener(IZonaListener) y removeListener(IZonaListener) para suscribir o desuscribir aquellas clases que estén interesadas en los cambios que se producen en la Zona.
Tiene dos mensajes para notificar sus posibles cambios de estado. Estos son notificarMuestraValidada(Muestra) y notificarNuevaMuestra(Muestra), que sirven para avisar a toda la lista de Listeners que en este caso serían todas las Organizaciones que implementen la interface IZonaListener y saben responder a los siguientes mensajes muestraRegistrada(Muestra,ZonaDeCobertura) y muestraValidada(Muestra,ZonaDeCobertura).



Clase Organización:

La clase Organización como mencionamos anteriormente aplica la interface IZonaListener pero además conoce a las Zonas que quiere suscribirse, y se encarga ella de su suscripción mediante el mensaje agregarZonaDeInteres(ZonaDeCobertura).
Cuando una Organización es notificada por un evento de alguna de sus zonas de interés. Avisa a alguno de sus colaboradores del tipo IFuncionalidadExterna, en este caso son PluginRegistro y PluginValidacion, que se encargan de ejecutar alguna acción específica dependiendo de si se valida o se registra una muestra. Estos colaboradores son llamados mediante los métodos de muestraRegistrada(Muestra,ZonaDeCobertura) o muestraValidada(Muestra,ZonaDeCobertura).